Final Results from the Tuning of the NMS_6b Weather Code

Daniel Pressel
US Army Research Laboratory (ARL), Aberdeen Proving Ground, MD
dmpresse@arl.army.mil

Abstract

The Nonhydrostatic Model Simulation (NMS) _6b weather code is commonly used by weather forecasters at television stations across the country. It is also of interest to the US Army due to its superior ability to predict weather in mountainous terrain. It was developed by Dr. Greg Tripoli of the University of Wisconsin[1]*. Currently this code exists only as a shared memory application parallelized using OpenMP. Attempts to parallelize it for distributed memory systems using MPI proved to be beyond the scope of the project that the author was involved in. This paper discusses efforts to improve the performance of the code at both the processor level and at the level of parallel performance.*

1. Introduction

Initial efforts to port and tune the NMS_6b code were discussed in Reference 2. These efforts included porting the NMS_6b weather code to the Cray X1, a single 32 processor node of the IBM SP with Power4 processors, and to parallelize the code using MPI instead of OpenMP. It was concluded that the program was not a good candidate for porting to the Cray X1 and this portion of the project was terminated. Similarly, it was concluded that efforts to use MPI were beyond the scope of this project due to complications involving support for moving nested grids. This paper discusses the remaining aspects associated with the optimization of both the serial and parallel (OpenMP) performance of the code. Issues involving the use of a single 32 processor node of an IBM SP with Power4 processors will be discussed. The remainder of this paper will cover the following topics:

- The effect that granularity of parallelization has on performance.
- Other impediments to achieving linear speedup.
- The way in which the code interacted with the memory system, in particular with the stream buffers.
- Cache effects.
- Translation Lookaside Buffers (TLBs).
- Methods used to reduce the time spent on integer arithmetic.
- Large page sizes

In addition to the approaches to tuning and parallelizing discussed in the remainder of the paper, a number of fairly standard techniques were used to improve the performance of this code. The majority of these techniques were used to eliminate redundant calculations (especially those involving the exponentiation operator). In general, the savings in any one routine was small, but the combined savings amounted to 15–20 minutes for a 6 hour run. When combined with the other optimizations discussed in this paper, the total improvement in run time was more than a factor of 2 for a run that had been taking over 9 hours.

This project was supported by a grant of computer time from the DoD High Performance Computing Modernization Program.

2. The Effect that Granularity of Parallelization has on Performance

Based on a variety of talks that the author has attended as well as some one-on-one discussions, it appears as though a commonly held misconception is that it is almost always a win to take advantage of any opportunities for parallelizing code one finds. Even when it is not a win, the general assumption appears to be that it is not much of a loss. As was discussed in Reference 3, this is most definitely not the case. A rough estimate of the cost of parallelizing a section of code using loop-level parallelism is to estimate the cost of synchronization.

For small numbers of processors and bus based systems of any size, this will be approximately equal to Memory Latency * C * N. C is a constant, and is most likely to be either 2 or 4 (we will assume 4). N is the number of processors. The memory latency is likely to be at least 200 nanosecond (NS), and on some systems may exceed 1000 NS. For purposes of discussion, we will assume 400 NS. Larger systems may use a more sophisticated algorithm for synchronization, in which case the cost would approximately equal Memory Latency * C * C * Log base 2(N), where C and N are defined as above. For 32 processors this gives a synchronization cost of 51 microseconds. For 256 processors, the synchronization cost assuming the more sophisticated algorithm would be at least 28 microseconds, and might be larger since large systems tend to have larger memory latencies.

From this discussion, one can conclude that to avoid parallel slowdown, a parallelized section of code should require not less than 28 microseconds to execute on a single processor. In order to show good speedup, the amount of work should be at least a factor of 100 * N greater than this. For 32 processors that comes to 0.16 seconds. For 256 processors, that comes to 0.72 seconds. The importance of this discussion is that the subroutine AZERO was frequently used to initialize arrays to zero. Over 99 percent of the time, these arrays were 84 elements long. Assuming that the arrays are already in the L1 cache, this could be done in roughly 84 cycles. On an SGI Origin 3000 with 400 MHz processors, this comes to 210 NS. On systems with newer processors, this value would be even less. If the array needs to be fetched from memory, and later written back to memory, and assuming a usable memory bandwidth of 500 MB/second and 4 byte data, the initialization would take 670 NS. Clearly, these calls should never have been parallelized. On the other hand, a small percentage of the calls involved considerably larger arrays and therefore those calls needed to be parallelized. Based on this analysis, AZERO was split into multiple loops, and depending on the size of the array, different code is selected for execution.

AZERO was not the only short count loop that had been parallelized. We believe that all such loops have now been identified and the parallelization constructs were either removed entirely or made conditional as in the case of AZERO. This had the unfortunate side effect of increasing the percentage of serial code. In other words, it effectively helps to establish an upper bound on the parallel speedup when using large numbers of processors. In an attempt to limit this side effect, ways were found to eliminate most of the calls to AZERO. Additionally, efforts were made to maximize the efficiency of AZERO and related routines. One of the most promising of these approaches appeared to be to use one or another of the compiler directives for data prefetching in an attempt to minimize the cost of initializing the large arrays (this was now over 50 percent of the time spent in AZERO). Unfortunately, it wasn't until the project was nearly completed that it was discovered that on the IBM SP these directives are normally ignored when they occur in loops parallelized using OpenMP. There appears to be a complicated set of compiler options that one can use to get around this limitation, but we ran out of time before we could make them work.

The initial parallelization effort left many loops to the automatic parallelization option found on most of today's compilers. So long as the loops were simple enough, this was an efficient division of labor. However, a careful analysis of the output from a profiled run indicated that some of these loops were not being parallelized. Individually, the cost of these loops was miniscule. Even when taken in combination, they represented a small percentage of the total CPU time. The problem was that when run on 30 processors, the combined time spent in these loops was 15–20 minutes for a 6 hour run. Some of the loops proved to be difficult to parallelize, but enough of the work was either parallelized or eliminated through serial optimizations to reduce the run time by at least 15 minutes.

3. Other Impediments to Achieving Linear Speedup

As is discussed in Reference 3, when dividing finite units of parallelism between processors, the ideal speedup will more closely resemble a staircase than a straight line (e.g., with 15 units of parallelism it can be divided evenly between 5 processors, but not 4 or 6 processors). This effect becomes most pronounced as the number of processors approaches the degree of available parallelism. In the case of NMS_6b, this effect is further complicated by the internal workings of the code. Based on certain hard coded parameters, the program clusters multiple units of parallelism into a single group. Effectively, this reduces the available level of parallelism. When absolutely necessary, one can change the hard coded parameters, thereby increasing the usable level of parallelism. Tests were run on the IBM SP with Power 4 processors to see what effect this would have on the run time. The hope was that as long as the parallelism was roughly an integer multiple of the number of processors being used, the run time would be unaffected. This was not what we found. Instead, the smaller the group size, the greater the usable level of parallelism, but also the greater the run time, by up to 25 %. The obvious solution for the IBM SP where we were limited to using no more than 32 processors was to use the largest group size possible. However, for systems with larger numbers of processors this effect would be one more impediment to achieving anything close to linear speedup.

Another problem has to do with competition between the operating system and the application for processor time. When dealing with a large system such as the SGI Origin 3000 with up to 2048 processors, it is reasonable to expect that a few processors will be reserved for use by the operating system, interactive jobs, and the like. On systems with smaller node counts, the normal practice is to schedule work on all of the processors. Therefore, with a 32 processor node size, one would normally expect the load factor to be 32. Adding in the operating system overhead, one will generally see a load factor of roughly 32.5. When dealing with a single 32 processor coarse grained application, the additional .5 units of work is easily spread across the 32 processors, resulting in a predicted parallel speedup of 31.5. However, NMS_6b is parallelized using loop level parallelism, which is an example of fine grained parallelism. Due to the increased frequency of synchronization events, even if the operating system overhead is spread across all 32 processors, it behaves as though it is running on a single processor. Therefore, the predicted speedup when using 32 processors would only be 16, while the predicted speedup when using 31 processors would be 31! Based on this analysis, measurements were made and it was found that for the test case there was little difference in performance between 30 and 31 processors (due to the stair step effect), but that 32 processor runs were noticeably slower. From that point on, all runs were made using 30 processors on a dedicated 32 processor node.

Another impediment to achieving parallel speedup on an SMP node is the available memory bandwidth on a per processor basis. On paper, the IBM SP with Power 4 processors has a very impressive memory bandwidth. However, we were not seeing anything close to the peak bandwidth. After checking with IBM, it was determined that the peak bandwidth for a node equipped with less than 128 GB of main memory is significantly less than the published value. The system that we were using was equipped with 32 GB of main memory per node. In and of itself, this would not appear to be an impediment to achieving linear speedup, however with prefetching a single processor should be able to take advantage of at least M/N MB/sec of memory bandwidth, where M is the peak memory bandwidth and N is the maximum number of processors in a node. Since the system being used only had a peak memory bandwidth of $m < M$, it is not surprising that one would run out of memory bandwidth before running out of processors. When dealing with a collection of serial jobs, this is not likely to be a serious problem, since different parts of a program will put different levels of stress on the memory system. Therefore, assuming that each job randomly enters and exits the high bandwidth and low bandwidth subroutines, then one might not see much of a problem. Similarly, most jobs parallelized using MPI are coarse grained jobs. While the progress of each process associated with the MPI job is far from random, there is enough of a disconnect that one might see this problem to a much smaller extent. Unfortunately, since NMS_6b is parallelized using loop-level parallelism, which is inherently fine grained, all of the processors progress in lock step between high bandwidth and low bandwidth subroutines. Therefore, even if on average there is enough bandwidth to show linear speedup, in practice the high bandwidth subroutines are likely to be bandwidth starved when all/most of the processors are being used by a single job. This appears to be the best explanation as to why some of the subroutines showed nearly linear speedup, while others showed at best factors of 20–25 speedups.

4. The Way in Which the Code Interacted with the Memory System, in Particular with the Stream Buffers

One key aspect of the IBM SP with Power 4 processors is the ability of the processor to stream data into the caches from main memory and on up to the processor. If each L3 cache miss was handled individually as it happened, most of the memory bandwidth would be unusable. Instead, the processor attempts to stream the data into the L3 processor whenever possible. While prefetch instructions can help to improve the efficiency of this process, the key component is the group of 8 stream buffers. This means that up to 8 separate data streams can be efficiently moved from main memory at one time. This raises the question, what happens if there are more than 8 data streams? The answer is not pretty. The performance of the section of code in question can drop off rapidly. The extent to which this drop off is seen will in part depend on what the overall cache miss rate is. The lower the miss rate, the less of a problem one is likely to see.

In the case of NMS_6b, one of the second most expensive subroutine had 20 arrays that it was accessing. Given the structure of this subroutine, this translated into at least 20 data streams. By carefully breaking the middle loop into multiple loops and reordering some of the inner loops, it was possible to reduce the maximum number of data streams in a middle loop to 8. The outer loop still had 20 data streams, but the stream buffers were primarily concerned with the inner and middle loops. As a result of this and other optimizations, the speed of this routine was doubled.

5. Cache Effects

An important aspect of the Power 4 processor is the L3 cache. Its latency is 90 NS. The system that was used in this exercise had a 1.3 GHz clock rate (.77 NS cycle time). This means that if an L3 cache hit stalls the processor without being overlapped with other useful work, the processor will lose the ability to execute 467 floating point operations. This is significantly more than the amount of lost work associated with L2 cache hits on many competing products. One might object to comparing an L3 cache to an L2 cache, however based on the size of the caches (16 MB for the L3 cache and 4-8 MB for many L2 caches), this seems to be the most appropriate comparison. This means that with the Power 4 processor, it is important to tune for both the L2 and L3 caches. Unfortunately, on a per processor basis, the L2 cache is less than 1MB in size.

As was previously mentioned, many of the scratch arrays are 84 elements in size. What was not mentioned is that only the first 35 elements of these arrays are being used. Attempts were made to decrease the size of these arrays, however for reasons that were not immediately clear, they broke the program. As a general rule, this would have been little more than a minor annoyance. However, when tuning loops with a STRIDE-N access pattern, one frequently uses blocking. A side effect is to change one dimensional arrays into two dimensional arrays. Given a 4 byte data size and a 128 byte cache line size for the L1 and L2 caches, on average one would expect 76 percent of the 2-D array to be brought into the L1 cache, even though only 42 percent of the elements were being used. A more serious problem was that in many cases the scratch array was no longer fitting in the L2 cache, or at best had a low hit rate due to a high rate of eviction. By moving the declaration of the scratch array to the parent subroutine, we were able to leave the amount of allocated space unchanged. However, inside of the routine where the array was actually being used, we declared the scratch array with the smaller dimensions. This approach was applied to many of the more expensive subroutines and resulted in an overall savings of approximately 10% of the run time.

6. Translation Lookaside Buffers (TLBs)

All RISC and CISC based systems divide main memory into pages. Similarly, the logical address space of a program is divided into pages. Page Table Entries (PTEs) are used to map the logical pages to the physical pages in main memory. To improve the performance of this process, a Translation Lookaside Buffer (TLB) is used to cache recently used PTEs either inside of, or at least close to the CPU. This puts an upper bound on the size of a TLB. Most systems use TLBs with 64–256 entries. The Power 4 processor is equipped with a 1024 entry TLB. Even so, when accessing multiple arrays in a STRIDE-N access pattern, one would not be surprised to find a large number of TLB misses. Using Perfex on an SGI Origin 3000 and HPMcount on an IBM SP with Power 4 processors, it was found that on both systems there were an extraordinarily large number of TLB misses. Several of the more expensive subroutines were spending 50–90% of their time on TLB misses on the IBM system.

The most desirable solution to this problem is to either reorder the indices of the arrays, or reorder the loops in a loop nest so that there is a STRIDE-1 access pattern. Many codes perform sweeps through the data first in one direction, and then in another direction. This is a critical part of the algorithm. In such a case, it will be impossible to eliminate all of the STRIDE-N access patterns. The best that can be done in such a situation is to block the code. This approach was used with many of the more expensive loops in NMS_6b. However, in a few cases, the use of GOTOs and the hard to follow logic made it difficult to apply this technique. In one such case, it required multiple attempts before the routine was successfully tuned without damaging the logic. Given the complexity of this effort several related routines that were called less frequently were left alone. Upon conclusion of this project, over 90% of all TLB misses were eliminated. This resulted in an overall savings of at least 20% of the run time.

An interesting side note is that one expensive loop had to be left alone. Attempts to apply blocking to this loop required converting 1-D scratch arrays into 2-D scratch arrays. The 1-D arrays had a high L1 cache hit rate, and a nearly 0% L2 cache miss rate. The 2-D scratch array had a poor L1 cache hit rate and a good, but not great, L2 cache hit rate. As a result of the increase number of L1 and L2 cache misses, the cost of the additional cache misses significantly exceeded the savings from the reduction in TLB misses on the IBM SP with Power 4 processors.

7. Methods Used to Reduce the Time Spent on Integer Arithmetic

A surprising finding when running HPMcount was that NMS_6b was spending a significant percentage of its time (probably in excess of 50% of the time, and more like 90% of the instructions) executing integer instructions. It was found that several of the most expensive loops in the most expensive subroutine were constantly testing in inner loops to see if the current location was above or below ground. This information was stored in a two dimensional integer array. In some but not all cases, it was possible to move this test either to an outer loop, or to move the test out of the body of the loop and into the DO statement itself. In the remaining cases, it was determined that no matter how mountainous the terrain is, the top of the atmosphere is significantly above ground level. Therefore, if one checks ahead of time for what the highest elevation for land (a 2-D calculation), one can then use that value to move most of the tests outside of the inner most loop. The resulting savings for the 3-D calculation more than offset the 2-D calculation.

A careful analysis of usage patterns allowed us to avoid creating arrays that were never being used for the specified input data (for other input data it may still be necessary to create the arrays). A selective application of techniques for software pipelining allowed the elimination of many scratch variables and redundant calculations. Overall, the cost of the most expensive subroutine was reduced by more than a factor of 2.

8. Large Page Sizes

Documentation from IBM recommends using large memory pages as a way of improving performance. In theory they should reduce the number of TLB misses. Furthermore, in the case of the Power 4 processor, they should improve the efficiency of the stream buffers. Early efforts to use large memory pages demonstrated their value. However, we were surprised to find that once the program had been tuned along the lines discussed in this paper, the use of large pages actually was counter productive. For reasons that are not clear, turning on that feature actually slowed the program down by a few percent.

9. Conclusions

The final result of this tuning effort was to reduce the run time to 4 hours. Ideally, further reductions in run time would have been highly desirable. Additional tuning might be of some value (e.g., 20–50 % reduction in run time). The new IBM SP Power 4+ with a 25% faster memory system and a 30 % faster processor might also help. However, it would appear as though the best approach would be to consider other shared memory systems. Some examples of these are:

- SGI Altix with up to 256 6 GFLOP processors.
- Fujitsu makes a SPARC based with up to 128 6 GFLOP processors
- HP makes the Super Dome and GS product lines.

Possibly one of these would allow us to reach the original goal of a 1 hour run with the least amount of additional effort.

References

1. Tripoli, G.J., "A Nonhydrostatic Mesoscale Model Designed to Simulated Scale Interaction." *Monthly Weather Review*, vol. 120, no. 7, 1992, pp. 1342–359.

2. Thompson, Steven R., et al., "Optimization of the NMS6b Weather Model Code." *Proceedings of the DoD HPCMP 2003 Users Group Conference*, June 2003.

3. Pressel, Daniel M. "The Scalability of Loop-Level Parallelism." *ARL-TR-2557*, published by the US Army Research Laboratory, August 2001.